java.util

Class Collections

java.lang.Object
    java.util.Collections

```
public class Collections
extends Object
```

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The methods of this class all throw a `NullPointerException` if the collections or class objects provided to them are null.

The documentation for the polymorphic algorithms contained in this class generally includes a brief description of the *implementation*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort` does not have to be a mergesort, but it does have to be *stable*.)

The "destructive" algorithms contained in this class, that is, the algorithms that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if the collection does not support the appropriate mutation primitive(s), such as the `set` method. These algorithms may, but are not required to, throw this exception if an invocation would have no effect on the collection. For example, invoking the `sort` method on an unmodifiable list that is already sorted may or may not throw `UnsupportedOperationException`.

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `Set`, `List`, `Map`

Field Summary

Fields

Modifier and Type	Field and Description
`static List`	**`EMPTY_LIST`** The empty list (immutable).
`static Map`	**`EMPTY_MAP`** The empty map (immutable).
`static Set`	**`EMPTY_SET`** The empty set (immutable).

Method Summary

Methods

Modifier and Type	Method and Description
static <T> boolean	**addAll**(**Collection**<? super T> c, T... elements) Adds all of the specified elements to the specified collection.
static <T> **Queue**<T>	**asLifoQueue**(**Deque**<T> deque) Returns a view of a **Deque** as a Last-in-first-out (Lifo) **Queue**.
static <T> int	**binarySearch**(**List**<? extends **Comparable**<? super T>> list, T key) Searches the specified list for the specified object using the binary search algorithm.
static <T> int	**binarySearch**(**List**<? extends T> list, T key, **Comparator**<? super T> c) Searches the specified list for the specified object using the binary search algorithm.
static <E> **Collection**<E>	**checkedCollection**(**Collection**<E> c, **Class**<E> type) Returns a dynamically typesafe view of the specified collection.
static <E> **List**<E>	**checkedList**(**List**<E> list, **Class**<E> type) Returns a dynamically typesafe view of the specified list.
static <K,V> **Map**<K,V>	**checkedMap**(**Map**<K,V> m, **Class**<K> keyType, **Class**<V> valueType) Returns a dynamically typesafe view of the specified map.
static <E> **Set**<E>	**checkedSet**(**Set**<E> s, **Class**<E> type) Returns a dynamically typesafe view of the specified set.
static <K,V> **SortedMap**<K,V>	**checkedSortedMap**(**SortedMap**<K,V> m, **Class**<K> keyType, **Class**<V> valueType) Returns a dynamically typesafe view of the specified sorted map.
static <E> **SortedSet**<E>	**checkedSortedSet**(**SortedSet**<E> s, **Class**<E> type) Returns a dynamically typesafe view of the specified sorted set.
static <T> void	**copy**(**List**<? super T> dest, **List**<? extends T> src) Copies all of the elements from one list into another.
static boolean	**disjoint**(**Collection**<?> c1, **Collection**<?> c2) Returns `true` if the two specified collections have no elements in common.
static <T> **Enumeration**<T>	**emptyEnumeration**() Returns an enumeration that has no elements.
static <T> **Iterator**<T>	**emptyIterator**() Returns an iterator that has no elements.
static <T> **List**<T>	**emptyList**() Returns the empty list (immutable).
static <T> **ListIterator**<T>	**emptyListIterator**() Returns a list iterator that has no elements.
static <K,V> **Map**<K,V>	**emptyMap**() Returns the empty map (immutable).
static <T> **Set**<T>	**emptySet**() Returns the empty set (immutable).

static <T> **Enumeration**<T>	**enumeration**(**Collection**<T> c) Returns an enumeration over the specified collection.
static <T> void	**fill**(**List**<? super T> list, T obj) Replaces all of the elements of the specified list with the specified element.
static int	**frequency**(**Collection**<?> c, **Object** o) Returns the number of elements in the specified collection equal to the specified object.
static int	**indexOfSubList**(**List**<?> source, **List**<?> target) Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static int	**lastIndexOfSubList**(**List**<?> source, **List**<?> target) Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static <T> **ArrayList**<T>	**list**(**Enumeration**<T> e) Returns an array list containing the elements returned by the specified enumeration in the order they are returned by the enumeration.
static <T extends **Object** & **Comparable**<? super T>> T	**max**(**Collection**<? extends T> coll) Returns the maximum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**max**(**Collection**<? extends T> coll, **Comparator**<? super T> comp) Returns the maximum element of the given collection, according to the order induced by the specified comparator.
static <T extends **Object** & **Comparable**<? super T>> T	**min**(**Collection**<? extends T> coll) Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**min**(**Collection**<? extends T> coll, **Comparator**<? super T> comp) Returns the minimum element of the given collection, according to the order induced by the specified comparator.
static <T> **List**<T>	**nCopies**(int n, T o) Returns an immutable list consisting of n copies of the specified object.
static <E> **Set**<E>	**newSetFromMap**(**Map**<E,**Boolean**> map) Returns a set backed by the specified map.
static <T> boolean	**replaceAll**(**List**<T> list, T oldVal, T newVal) Replaces all occurrences of one specified value in a list with another.
static void	**reverse**(**List**<?> list) Reverses the order of the elements in the specified list.
static <T> **Comparator**<T>	**reverseOrder**() Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the `Comparable` interface.
static <T> **Comparator**<T>	**reverseOrder**(**Comparator**<T> cmp) Returns a comparator that imposes the reverse ordering of the specified comparator.

static void	**rotate**(List<?> list, int distance) Rotates the elements in the specified list by the specified distance.
static void	**shuffle**(List<?> list) Randomly permutes the specified list using a default source of randomness.
static void	**shuffle**(List<?> list, Random rnd) Randomly permute the specified list using the specified source of randomness.
static <T> Set<T>	**singleton**(T o) Returns an immutable set containing only the specified object.
static <T> List<T>	**singletonList**(T o) Returns an immutable list containing only the specified object.
static <K,V> Map<K,V>	**singletonMap**(K key, V value) Returns an immutable map, mapping only the specified key to the specified value.
static <T extends Comparable<? super T>> void	**sort**(List<T> list) Sorts the specified list into ascending order, according to the **natural ordering** of its elements.
static <T> void	**sort**(List<T> list, Comparator<? super T> c) Sorts the specified list according to the order induced by the specified comparator.
static void	**swap**(List<?> list, int i, int j) Swaps the elements at the specified positions in the specified list.
static <T> Collection<T>	**synchronizedCollection**(Collection<T> c) Returns a synchronized (thread-safe) collection backed by the specified collection.
static <T> List<T>	**synchronizedList**(List<T> list) Returns a synchronized (thread-safe) list backed by the specified list.
static <K,V> Map<K,V>	**synchronizedMap**(Map<K,V> m) Returns a synchronized (thread-safe) map backed by the specified map.
static <T> Set<T>	**synchronizedSet**(Set<T> s) Returns a synchronized (thread-safe) set backed by the specified set.
static <K,V> SortedMap<K,V>	**synchronizedSortedMap**(SortedMap<K,V> m) Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
static <T> SortedSet<T>	**synchronizedSortedSet**(SortedSet<T> s) Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.
static <T> Collection<T>	**unmodifiableCollection**(Collection<? extends T> c) Returns an unmodifiable view of the specified collection.
static <T> List<T>	**unmodifiableList**(List<? extends T> list) Returns an unmodifiable view of the specified list.
static <K,V> Map<K,V>	**unmodifiableMap**(Map<? extends K,? extends V> m) Returns an unmodifiable view of the specified map.
static <T> Set<T>	**unmodifiableSet**(Set<? extends T> s) Returns an unmodifiable view of the specified set.

static <K,V> **SortedMap**<K,V>	**unmodifiableSortedMap**(**SortedMap**<K,? extends V> m) Returns an unmodifiable view of the specified sorted map.
static <T> **SortedSet**<T>	**unmodifiableSortedSet**(**SortedSet**<T> s) Returns an unmodifiable view of the specified sorted set.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

EMPTY_SET

```
public static final Set EMPTY_SET
```

The empty set (immutable). This set is serializable.

See Also:

emptySet()

EMPTY_LIST

```
public static final List EMPTY_LIST
```

The empty list (immutable). This list is serializable.

See Also:

emptyList()

EMPTY_MAP

```
public static final Map EMPTY_MAP
```

The empty map (immutable). This map is serializable.

Since:

1.3

See Also:

emptyMap()

Method Detail

sort

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must

implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the $n^2 \log(n)$ performance that would result from attempting to sort a linked list in place.

Parameters:

`list` - the list to be sorted.

Throws:

`ClassCastException` - if the list contains elements that are not *mutually comparable* (for example, strings and integers).

`UnsupportedOperationException` - if the specified list's list-iterator does not support the `set` operation.

`IllegalArgumentException` - (optional) if the implementation detects that the natural ordering of the list elements is found to violate the `Comparable` contract

sort

```
public static <T> void sort(List<T> list,
            Comparator<? super T> c)
```

Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be *mutually comparable* using the specified comparator (that is, `c.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the $n^2 \log(n)$ performance that would result from attempting to sort a linked list in place.

Parameters:

`list` - the list to be sorted.

c - the comparator to determine the order of the list. A `null` value indicates that the elements' *natural ordering* should be used.

Throws:

`ClassCastException` - if the list contains elements that are not *mutually comparable* using the specified comparator.

`UnsupportedOperationException` - if the specified list's list-iterator does not support the `set` operation.

`IllegalArgumentException` - (optional) if the comparator is found to violate the `Comparator` contract

binarySearch

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list,
                   T key)
```

Searches the specified list for the specified object using the binary search algorithm. The list must be sorted into ascending order according to the natural ordering of its elements (as by the `sort(List)` method) prior to making this call. If it is not sorted, the results are undefined. If the list contains multiple elements equal to the specified object, there is no guarantee which one will be found.

This method runs in log(n) time for a "random access" list (which provides near-constant-time positional access). If the specified list does not implement the `RandomAccess` interface and is large, this method will do an iterator-based binary search that performs O(n) link traversals and O(log n) element comparisons.

Parameters:

`list` - the list to be searched.

key - the key to be searched for.

Returns:

the index of the search key, if it is contained in the list; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or `list.size()` if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`ClassCastException` - if the list contains elements that are not *mutually comparable* (for example, strings and integers), or the search key is not mutually comparable with the elements of the list.

binarySearch

```
public static <T> int binarySearch(List<? extends T> list,
                   T key,
                   Comparator<? super T> c)
```

Searches the specified list for the specified object using the binary search algorithm. The list must be sorted into ascending order according to the specified comparator (as by the `sort(List, Comparator)` method), prior to making this call. If it is not sorted, the results are undefined. If the list contains multiple elements equal to the specified object, there is no guarantee which one will be found.

This method runs in log(n) time for a "random access" list (which provides near-constant-time positional access). If the specified list does not implement the `RandomAccess` interface and is large, this method will do an iterator-based binary search that performs O(n) link traversals and O(log n) element comparisons.

Parameters:

`list` - the list to be searched.

key - the key to be searched for.

`c` - the comparator by which the list is ordered. A `null` value indicates that the elements' natural ordering should be used.

Returns:

the index of the search key, if it is contained in the list; otherwise, `(-(insertion point) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or `list.size()` if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`ClassCastException` - if the list contains elements that are not *mutually comparable* using the specified comparator, or the search key is not mutually comparable with the elements of the list using this comparator.

reverse

```
public static void reverse(List<?> list)
```

Reverses the order of the elements in the specified list.

This method runs in linear time.

Parameters:

`list` - the list whose elements are to be reversed.

Throws:

`UnsupportedOperationException` - if the specified list or its list-iterator does not support the `set` operation.

shuffle

```
public static void shuffle(List<?> list)
```

Randomly permutes the specified list using a default source of randomness. All permutations occur with approximately equal likelihood.

The hedge "approximately" is used in the foregoing description because default source of randomness is only approximately an unbiased source of independently chosen bits. If it were a perfect source of randomly chosen bits, then the algorithm would choose permutations with perfect uniformity.

This implementation traverses the list backwards, from the last element up to the second, repeatedly swapping a randomly selected element into the "current position". Elements are randomly selected from the portion of the list that runs from the first element to the current position, inclusive.

This method runs in linear time. If the specified list does not implement the `RandomAccess` interface and is large, this implementation dumps the specified list into an array before shuffling it, and dumps the shuffled array back into the list. This avoids the quadratic behavior that would result from shuffling a "sequential access" list in place.

Parameters:

`list` - the list to be shuffled.

Throws:

`UnsupportedOperationException` - if the specified list or its list-iterator does not support the `set` operation.

shuffle

```
public static void shuffle(List<?> list,
           Random rnd)
```

Randomly permute the specified list using the specified source of randomness. All permutations occur with equal likelihood assuming that the source of randomness is fair.

This implementation traverses the list backwards, from the last element up to the second, repeatedly swapping a randomly selected element into the "current position". Elements are randomly selected from the portion of the list that runs from the first element to the current position, inclusive.

This method runs in linear time. If the specified list does not implement the `RandomAccess` interface and is large, this implementation dumps the specified list into an array before shuffling it, and dumps the shuffled array back into the list. This avoids the quadratic behavior that would result from shuffling a "sequential access" list in place.

Parameters:

`list` - the list to be shuffled.

`rnd` - the source of randomness to use to shuffle the list.

Throws:

`UnsupportedOperationException` - if the specified list or its list-iterator does not support the `set` operation.

swap

```
public static void swap(List<?> list,
        int i,
        int j)
```

Swaps the elements at the specified positions in the specified list. (If the specified positions are equal, invoking this method leaves the list unchanged.)

Parameters:

`list` - The list in which to swap elements.

`i` - the index of one element to be swapped.

`j` - the index of the other element to be swapped.

Throws:

`IndexOutOfBoundsException` - if either `i` or `j` is out of range (i < 0 || i >= list.size() || j < 0 || j >= list.size()).

Since:

1.4

fill

```
public static <T> void fill(List<? super T> list,
            T obj)
```

Replaces all of the elements of the specified list with the specified element.

This method runs in linear time.

Parameters:

`list` - the list to be filled with the specified element.

`obj` - The element with which to fill the specified list.

Throws:

`UnsupportedOperationException` - if the specified list or its list-iterator does not support the `set` operation.

copy

```
public static <T> void copy(List<? super T> dest,
            List<? extends T> src)
```

Copies all of the elements from one list into another. After the operation, the index of each copied element in the destination list will be identical to its index in the source list. The destination list must be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected.

This method runs in linear time.

Parameters:

`dest` - The destination list.

`src` - The source list.

Throws:

`IndexOutOfBoundsException` - if the destination list is too small to contain the entire source List.

`UnsupportedOperationException` - if the destination list's list-iterator does not support the `set` operation.

min

```
public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)
```

Returns the minimum element of the given collection, according to the *natural ordering* of its elements. All elements in the collection must implement the `Comparable` interface. Furthermore, all elements in the collection must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the collection).

This method iterates over the entire collection, hence it requires time proportional to the size of the collection.

Parameters:

`coll` - the collection whose minimum element is to be determined.

Returns:

the minimum element of the given collection, according to the *natural ordering* of its elements.

Throws:

`ClassCastException` - if the collection contains elements that are not *mutually comparable* (for example, strings and integers).

`NoSuchElementException` - if the collection is empty.

See Also:

`Comparable`

min

```
public static <T> T min(Collection<? extends T> coll,
        Comparator<? super T> comp)
```

Returns the minimum element of the given collection, according to the order induced by the specified comparator. All elements in the collection must be *mutually comparable* by the specified comparator (that is, `comp.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the collection).

This method iterates over the entire collection, hence it requires time proportional to the size of the collection.

Parameters:

`coll` - the collection whose minimum element is to be determined.

`comp` - the comparator with which to determine the minimum element. A `null` value indicates that the elements' *natural ordering* should be used.

Returns:

the minimum element of the given collection, according to the specified comparator.

Throws:

`ClassCastException` - if the collection contains elements that are not *mutually comparable* using the specified comparator.

`NoSuchElementException` - if the collection is empty.

See Also:

`Comparable`

max

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

Returns the maximum element of the given collection, according to the *natural ordering* of its elements. All elements in the collection must implement the `Comparable` interface. Furthermore, all elements in the collection must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the collection).

This method iterates over the entire collection, hence it requires time proportional to the size of the collection.

Parameters:

`coll` - the collection whose maximum element is to be determined.

Returns:

the maximum element of the given collection, according to the *natural ordering* of its elements.

Throws:

`ClassCastException` - if the collection contains elements that are not *mutually comparable* (for example, strings and integers).

`NoSuchElementException` - if the collection is empty.

See Also:

`Comparable`

max

```
public static <T> T max(Collection<? extends T> coll,
        Comparator<? super T> comp)
```

Returns the maximum element of the given collection, according to the order induced by the specified comparator. All elements in the collection must be *mutually comparable* by the specified comparator (that is, `comp.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the collection).

This method iterates over the entire collection, hence it requires time proportional to the size of the collection.

Parameters:

`coll` - the collection whose maximum element is to be determined.

`comp` - the comparator with which to determine the maximum element. A `null` value indicates that the elements' *natural ordering* should be used.

Returns:

the maximum element of the given collection, according to the specified comparator.

Throws:

`ClassCastException` - if the collection contains elements that are not *mutually comparable* using the specified comparator.

`NoSuchElementException` - if the collection is empty.

See Also:

`Comparable`

rotate

```
public static void rotate(List<?> list,
          int distance)
```

Rotates the elements in the specified list by the specified distance. After calling this method, the element at index `i` will be the element previously at index `(i - distance)` mod `list.size()`, for all values of `i` between `0` and `list.size()-1`, inclusive. (This method has no effect on the size of the list.)

For example, suppose `list` comprises `[t, a, n, k, s]`. After invoking `Collections.rotate(list, 1)` (or `Collections.rotate(list, -4)`), `list` will comprise `[s, t, a, n, k]`.

Note that this method can usefully be applied to sublists to move one or more elements within a list while preserving the order of the remaining elements. For example, the following idiom moves the element at index `j` forward to position `k` (which must be greater than or equal to `j`):

```
Collections.rotate(list.subList(j, k+1), -1);
```

To make this concrete, suppose `list` comprises `[a, b, c, d, e]`. To move the element at index 1 (`b`) forward two positions, perform the following invocation:

```
Collections.rotate(l.subList(1, 4), -1);
```

The resulting list is `[a, c, d, b, e]`.

To move more than one element forward, increase the absolute value of the rotation distance. To move elements backward, use a positive shift distance.

If the specified list is small or implements the `RandomAccess` interface, this implementation exchanges the first element into the location it should go, and then repeatedly exchanges the displaced element into the location it should go until a displaced element is swapped into the first element. If necessary, the process is repeated on the second and successive elements, until the rotation is complete. If the specified list is large and doesn't implement the `RandomAccess` interface, this implementation breaks the list into two sublist views around index `-distance mod size`. Then the `reverse(List)` method is invoked on each sublist view, and finally it is invoked on the entire list. For a more complete description of both algorithms, see Section 2.3 of Jon Bentley's *Programming Pearls* (Addison-Wesley, 1986).

Parameters:

`list` - the list to be rotated.

`distance` - the distance to rotate the list. There are no constraints on this value; it may be zero, negative, or greater than `list.size()`.

Throws:

`UnsupportedOperationException` - if the specified list or its list-iterator does not support the `set` operation.

Since:

1.4

replaceAll

```
public static <T> boolean replaceAll(List<T> list,
                     T oldVal,
                     T newVal)
```

Replaces all occurrences of one specified value in a list with another. More formally, replaces with `newVal` each element `e` in `list` such that `(oldVal==null ? e==null : oldVal.equals(e))`. (This method has no effect on the size of the list.)

Parameters:

`list` - the list in which replacement is to occur.

`oldVal` - the old value to be replaced.

`newVal` - the new value with which `oldVal` is to be replaced.

Returns:

`true` if `list` contained one or more elements e such that `(oldVal==null ? e==null : oldVal.equals(e))`.

Throws:

`UnsupportedOperationException` - if the specified list or its list-iterator does not support the `set` operation.

Since:

1.4

indexOfSubList

```
public static int indexOfSubList(List<?> source,
                 List<?> target)
```

Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence. More formally, returns the lowest index `i` such that `source.subList(i, i+target.size()).equals(target)`, or -1 if there is no such index. (Returns -1 if `target.size() > source.size()`.)

This implementation uses the "brute force" technique of scanning over the source list, looking for a match with the target at each location in turn.

Parameters:

`source` - the list in which to search for the first occurrence of `target`.

`target` - the list to search for as a subList of `source`.

Returns:

the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.

Since:

1.4

lastIndexOfSubList

```
public static int lastIndexOfSubList(List<?> source,
                     List<?> target)
```

Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence. More formally, returns the highest index `i` such that `source.subList(i, i+target.size()).equals(target)`, or -1 if there is no such index. (Returns -1 if `target.size() > source.size()`.)

This implementation uses the "brute force" technique of iterating over the source list, looking for a match with the target at each location in turn.

Parameters:

`source` - the list in which to search for the last occurrence of `target`.

`target` - the list to search for as a subList of `source`.

Returns:

the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.

Since:

1.4

unmodifiableCollection

```
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c)
```

Returns an unmodifiable view of the specified collection. This method allows modules to provide users with "read-only" access to internal collections. Query operations on the returned collection "read through" to the specified collection, and attempts to modify the returned collection, whether direct or via its iterator, result in an `UnsupportedOperationException`.

The returned collection does *not* pass the hashCode and equals operations through to the backing collection, but relies on `Object`'s `equals` and `hashCode` methods. This is necessary to preserve the contracts of these operations in the case that the backing collection is a set or a list.

The returned collection will be serializable if the specified collection is serializable.

Parameters:

c - the collection for which an unmodifiable view is to be returned.

Returns:

an unmodifiable view of the specified collection.

unmodifiableSet

```
public static <T> Set<T> unmodifiableSet(Set<? extends T> s)
```

Returns an unmodifiable view of the specified set. This method allows modules to provide users with "read-only" access to internal sets. Query operations on the returned set "read through" to the specified set, and attempts to modify the returned set, whether direct or via its iterator, result in an `UnsupportedOperationException`.

The returned set will be serializable if the specified set is serializable.

Parameters:

s - the set for which an unmodifiable view is to be returned.

Returns:

an unmodifiable view of the specified set.

unmodifiableSortedSet

```
public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)
```

Returns an unmodifiable view of the specified sorted set. This method allows modules to provide users with "read-only" access to internal sorted sets. Query operations on the returned sorted set "read through" to the specified sorted set. Attempts to modify the returned sorted set, whether direct, via its iterator, or via its `subSet`, `headSet`, or `tailSet` views, result in an `UnsupportedOperationException`.

The returned sorted set will be serializable if the specified sorted set is serializable.

Parameters:

s - the sorted set for which an unmodifiable view is to be returned.

Returns:

an unmodifiable view of the specified sorted set.

unmodifiableList

```
public static <T> List<T> unmodifiableList(List<? extends T> list)
```

Returns an unmodifiable view of the specified list. This method allows modules to provide users with "read-only" access to internal lists. Query operations on the returned list "read through" to the specified list, and attempts to modify the returned list, whether direct or via its iterator, result in an `UnsupportedOperationException`.

The returned list will be serializable if the specified list is serializable. Similarly, the returned list will implement `RandomAccess` if the specified list does.

Parameters:

`list` - the list for which an unmodifiable view is to be returned.

Returns:

an unmodifiable view of the specified list.

unmodifiableMap

```
public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)
```

Returns an unmodifiable view of the specified map. This method allows modules to provide users with "read-only" access to internal maps. Query operations on the returned map "read through" to the specified map, and attempts to modify the returned map, whether direct or via its collection views, result in an `UnsupportedOperationException`.

The returned map will be serializable if the specified map is serializable.

Parameters:

`m` - the map for which an unmodifiable view is to be returned.

Returns:

an unmodifiable view of the specified map.

unmodifiableSortedMap

```
public static <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,? extends V> m)
```

Returns an unmodifiable view of the specified sorted map. This method allows modules to provide users with "read-only" access to internal sorted maps. Query operations on the returned sorted map "read through" to the specified sorted map. Attempts to modify the returned sorted map, whether direct, via its collection views, or via its `subMap`, `headMap`, or `tailMap` views, result in an `UnsupportedOperationException`.

The returned sorted map will be serializable if the specified sorted map is serializable.

Parameters:

`m` - the sorted map for which an unmodifiable view is to be returned.

Returns:

an unmodifiable view of the specified sorted map.

synchronizedCollection

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c)
```

Returns a synchronized (thread-safe) collection backed by the specified collection. In order to guarantee serial access, it is critical that **all** access to the backing collection is accomplished through the returned collection.

It is imperative that the user manually synchronize on the returned collection when iterating over it:

```
Collection c = Collections.synchronizedCollection(myCollection);
   ...
synchronized (c) {
    Iterator i = c.iterator(); // Must be in the synchronized block
    while (i.hasNext())
       foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.

The returned collection does *not* pass the `hashCode` and `equals` operations through to the backing collection, but relies on `Object`'s equals and hashCode methods. This is necessary to preserve the contracts of these operations in the case that the backing collection is a set or a list.

The returned collection will be serializable if the specified collection is serializable.

Parameters:

c - the collection to be "wrapped" in a synchronized collection.

Returns:

a synchronized view of the specified collection.

synchronizedSet

```
public static <T> Set<T> synchronizedSet(Set<T> s)
```

Returns a synchronized (thread-safe) set backed by the specified set. In order to guarantee serial access, it is critical that **all** access to the backing set is accomplished through the returned set.

It is imperative that the user manually synchronize on the returned set when iterating over it:

```
Set s = Collections.synchronizedSet(new HashSet());
    ...
synchronized (s) {
    Iterator i = s.iterator(); // Must be in the synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.

The returned set will be serializable if the specified set is serializable.

Parameters:

s - the set to be "wrapped" in a synchronized set.

Returns:

a synchronized view of the specified set.

synchronizedSortedSet

```
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
```

Returns a synchronized (thread-safe) sorted set backed by the specified sorted set. In order to guarantee serial access, it is critical that **all** access to the backing sorted set is accomplished through the returned sorted set (or its views).

It is imperative that the user manually synchronize on the returned sorted set when iterating over it or any of its `subSet`, `headSet`, or `tailSet` views.

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet());
```

```
    ...
  synchronized (s) {
      Iterator i = s.iterator(); // Must be in the synchronized block
      while (i.hasNext())
          foo(i.next());
  }
```

or:

```
  SortedSet s = Collections.synchronizedSortedSet(new TreeSet());
  SortedSet s2 = s.headSet(foo);
      ...
  synchronized (s) {  // Note: s, not s2!!!
      Iterator i = s2.iterator(); // Must be in the synchronized block
      while (i.hasNext())
          foo(i.next());
  }
```

Failure to follow this advice may result in non-deterministic behavior.

The returned sorted set will be serializable if the specified sorted set is serializable.

Parameters:

s - the sorted set to be "wrapped" in a synchronized sorted set.

Returns:

a synchronized view of the specified sorted set.

synchronizedList

```
public static <T> List<T> synchronizedList(List<T> list)
```

Returns a synchronized (thread-safe) list backed by the specified list. In order to guarantee serial access, it is critical that **all** access to the backing list is accomplished through the returned list.

It is imperative that the user manually synchronize on the returned list when iterating over it:

```
  List list = Collections.synchronizedList(new ArrayList());
      ...
  synchronized (list) {
      Iterator i = list.iterator(); // Must be in synchronized block
      while (i.hasNext())
          foo(i.next());
  }
```

Failure to follow this advice may result in non-deterministic behavior.

The returned list will be serializable if the specified list is serializable.

Parameters:

`list` - the list to be "wrapped" in a synchronized list.

Returns:

a synchronized view of the specified list.

synchronizedMap

```
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
```

Returns a synchronized (thread-safe) map backed by the specified map. In order to guarantee serial access, it is critical that **all** access to the backing map is accomplished through the returned map.

It is imperative that the user manually synchronize on the returned map when iterating over any of its collection views:

```
Map m = Collections.synchronizedMap(new HashMap());
    ...
Set s = m.keySet();  // Needn't be in synchronized block
    ...
synchronized (m) {  // Synchronizing on m, not s!
    Iterator i = s.iterator(); // Must be in synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.

The returned map will be serializable if the specified map is serializable.

Parameters:

`m` - the map to be "wrapped" in a synchronized map.

Returns:

a synchronized view of the specified map.

synchronizedSortedMap

```
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)
```

Returns a synchronized (thread-safe) sorted map backed by the specified sorted map. In order to guarantee serial access, it is critical that **all** access to the backing sorted map is accomplished through the returned sorted map (or its views).

It is imperative that the user manually synchronize on the returned sorted map when iterating over any of its collection views, or the collections views of any of its `subMap`, `headMap` or `tailMap` views.

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap());
    ...
Set s = m.keySet();  // Needn't be in synchronized block
    ...
synchronized (m) {  // Synchronizing on m, not s!
    Iterator i = s.iterator(); // Must be in synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

or:

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap());
SortedMap m2 = m.subMap(foo, bar);
    ...
Set s2 = m2.keySet();  // Needn't be in synchronized block
    ...
synchronized (m) {  // Synchronizing on m, not m2 or s2!
    Iterator i = s.iterator(); // Must be in synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.

The returned sorted map will be serializable if the specified sorted map is serializable.

Parameters:

`m` - the sorted map to be "wrapped" in a synchronized sorted map.

Returns:

a synchronized view of the specified sorted map.

checkedCollection

```
public static <E> Collection<E> checkedCollection(Collection<E> c,
                                  Class<E> type)
```

Returns a dynamically typesafe view of the specified collection. Any attempt to insert an element of the wrong type will result in an immediate `ClassCastException`. Assuming a collection contains no incorrectly typed elements prior to the time a dynamically typesafe view is generated, and that all subsequent access to the collection takes place through the view, it is *guaranteed* that the collection cannot contain an incorrectly typed element.

The generics mechanism in the language provides compile-time (static) type checking, but it is possible to defeat this mechanism with unchecked casts. Usually this is not a problem, as the compiler issues warnings on all such unchecked operations. There are, however, times when static type checking alone is not sufficient. For example, suppose a collection is passed to a third-party library and it is imperative that the library code not corrupt the collection by inserting an element of the wrong type.

Another use of dynamically typesafe views is debugging. Suppose a program fails with a `ClassCastException`, indicating that an incorrectly typed element was put into a parameterized collection. Unfortunately, the exception can occur at any time after the erroneous element is inserted, so it typically provides little or no information as to the real source of the problem. If the problem is reproducible, one can quickly determine its source by temporarily modifying the program to wrap the collection with a dynamically typesafe view. For example, this declaration:

```
Collection<String> c = new HashSet<String>();
```

may be replaced temporarily by this one:

```
Collection<String> c = Collections.checkedCollection(
        new HashSet<String>(), String.class);
```

Running the program again will cause it to fail at the point where an incorrectly typed element is inserted into the collection, clearly identifying the source of the problem. Once the problem is fixed, the modified declaration may be reverted back to the original.

The returned collection does *not* pass the hashCode and equals operations through to the backing collection, but relies on `Object`'s `equals` and `hashCode` methods. This is necessary to preserve the contracts of these operations in the case that the backing collection is a set or a list.

The returned collection will be serializable if the specified collection is serializable.

Since `null` is considered to be a value of any reference type, the returned collection permits insertion of null elements whenever the backing collection does.

Parameters:

`c` - the collection for which a dynamically typesafe view is to be returned

`type` - the type of element that `c` is permitted to hold

Returns:

a dynamically typesafe view of the specified collection

Since:

1.5

checkedSet

```
public static <E> Set<E> checkedSet(Set<E> s,
                    Class<E> type)
```

Returns a dynamically typesafe view of the specified set. Any attempt to insert an element of the wrong type will result in an immediate `ClassCastException`. Assuming a set contains no incorrectly typed elements prior to the time a dynamically typesafe view is generated, and that all subsequent access to the set takes place through the view, it is *guaranteed* that the set cannot contain an incorrectly typed element.

A discussion of the use of dynamically typesafe views may be found in the documentation for the `checkedCollection` method.

The returned set will be serializable if the specified set is serializable.

Since `null` is considered to be a value of any reference type, the returned set permits insertion of null elements whenever the backing set does.

Parameters:

`s` - the set for which a dynamically typesafe view is to be returned

`type` - the type of element that `s` is permitted to hold

Returns:

a dynamically typesafe view of the specified set

Since:

1.5

checkedSortedSet

```
public static <E> SortedSet<E> checkedSortedSet(SortedSet<E> s,
                                Class<E> type)
```

Returns a dynamically typesafe view of the specified sorted set. Any attempt to insert an element of the wrong type will result in an immediate `ClassCastException`. Assuming a sorted set contains no incorrectly typed elements prior to the time a dynamically typesafe view is generated, and that all subsequent access to the sorted set takes place through the view, it is *guaranteed* that the sorted set cannot contain an incorrectly typed element.

A discussion of the use of dynamically typesafe views may be found in the documentation for the `checkedCollection` method.

The returned sorted set will be serializable if the specified sorted set is serializable.

Since `null` is considered to be a value of any reference type, the returned sorted set permits insertion of null elements whenever the backing sorted set does.

Parameters:

`s` - the sorted set for which a dynamically typesafe view is to be returned

`type` - the type of element that `s` is permitted to hold

Returns:

a dynamically typesafe view of the specified sorted set

Since:

1.5

checkedList

```
public static <E> List<E> checkedList(List<E> list,
                      Class<E> type)
```

Returns a dynamically typesafe view of the specified list. Any attempt to insert an element of the wrong type will result in an immediate `ClassCastException`. Assuming a list contains no incorrectly typed elements prior to the time a dynamically typesafe view is generated, and that all subsequent access to the list takes place through the view, it is *guaranteed* that the list cannot contain an incorrectly typed element.

A discussion of the use of dynamically typesafe views may be found in the documentation for the `checkedCollection` method.

The returned list will be serializable if the specified list is serializable.

Since `null` is considered to be a value of any reference type, the returned list permits insertion of null elements whenever the backing list does.

Parameters:

`list` - the list for which a dynamically typesafe view is to be returned

`type` - the type of element that `list` is permitted to hold

Returns:

a dynamically typesafe view of the specified list

Since:

1.5

checkedMap

```
public static <K,V> Map<K,V> checkedMap(Map<K,V> m,
                        Class<K> keyType,
                        Class<V> valueType)
```

Returns a dynamically typesafe view of the specified map. Any attempt to insert a mapping whose key or value have the wrong type will result in an immediate `ClassCastException`. Similarly, any attempt to modify the value currently associated with a key will result in an immediate `ClassCastException`, whether the modification is attempted directly through the map itself, or through a `Map.Entry` instance obtained from the map's `entry set` view.

Assuming a map contains no incorrectly typed keys or values prior to the time a dynamically typesafe view is generated, and that all subsequent access to the map takes place through the view (or one of its collection views), it is *guaranteed* that the map cannot contain an incorrectly typed key or value.

A discussion of the use of dynamically typesafe views may be found in the documentation for the `checkedCollection` method.

The returned map will be serializable if the specified map is serializable.

Since `null` is considered to be a value of any reference type, the returned map permits insertion of null keys or values whenever the backing map does.

Parameters:

`m` - the map for which a dynamically typesafe view is to be returned

`keyType` - the type of key that `m` is permitted to hold

`valueType` - the type of value that `m` is permitted to hold

Returns:

a dynamically typesafe view of the specified map

Since:

1.5

checkedSortedMap

```
public static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K,V> m,
                                    Class<K> keyType,
                                    Class<V> valueType)
```

Returns a dynamically typesafe view of the specified sorted map. Any attempt to insert a mapping whose key or value have the wrong type will result in an immediate `ClassCastException`. Similarly, any attempt to modify the value currently associated with a key will result in an immediate `ClassCastException`, whether the modification is attempted directly through the map itself, or through a `Map.Entry` instance obtained from the map's `entry set` view.

Assuming a map contains no incorrectly typed keys or values prior to the time a dynamically typesafe view is generated, and that all subsequent access to the map takes place through the view (or one of its collection views), it is *guaranteed* that the map cannot contain an incorrectly typed key or value.

A discussion of the use of dynamically typesafe views may be found in the documentation for the `checkedCollection` method.

The returned map will be serializable if the specified map is serializable.

Since `null` is considered to be a value of any reference type, the returned map permits insertion of null keys or values whenever the backing map does.

Parameters:

`m` - the map for which a dynamically typesafe view is to be returned

`keyType` - the type of key that `m` is permitted to hold

`valueType` - the type of value that `m` is permitted to hold

Returns:

a dynamically typesafe view of the specified map

Since:

1.5

emptyIterator

```
public static <T> Iterator<T> emptyIterator()
```

Returns an iterator that has no elements. More precisely,

- `hasNext` always returns `false`.
- `next` always throws `NoSuchElementException`.
- `remove` always throws `IllegalStateException`.

Implementations of this method are permitted, but not required, to return the same object from multiple invocations.

Returns:

an empty iterator

Since:

1.7

emptyListIterator

```
public static <T> ListIterator<T> emptyListIterator()
```

Returns a list iterator that has no elements. More precisely,

- `hasNext` and `hasPrevious` always return `false`.
- `next` and `previous` always throw `NoSuchElementException`.
- `remove` and `set` always throw `IllegalStateException`.
- `add` always throws `UnsupportedOperationException`.
- `nextIndex` always returns `0` .
- `previousIndex` always returns −1.

Implementations of this method are permitted, but not required, to return the same object from multiple invocations.

Returns:

an empty list iterator

Since:

1.7

emptyEnumeration

```
public static <T> Enumeration<T> emptyEnumeration()
```

Returns an enumeration that has no elements. More precisely,

- `hasMoreElements` always returns `false`.
- `nextElement` always throws `NoSuchElementException`.

Implementations of this method are permitted, but not required, to return the same object from multiple invocations.

Returns:

an empty enumeration

Since:

1.7

emptySet

```
public static final <T> Set<T> emptySet()
```

Returns the empty set (immutable). This set is serializable. Unlike the like-named field, this method is parameterized.

This example illustrates the type-safe way to obtain an empty set:

```
Set<String> s = Collections.emptySet();
```

Implementation note: Implementations of this method need not create a separate `Set` object for each call. Using this method is likely to have comparable cost to using the like-named field. (Unlike this method, the field does not provide type safety.)

Since:

1.5

See Also:

`EMPTY_SET`

emptyList

```
public static final <T> List<T> emptyList()
```

Returns the empty list (immutable). This list is serializable.

This example illustrates the type-safe way to obtain an empty list:

```
List<String> s = Collections.emptyList();
```

Implementation note: Implementations of this method need not create a separate `List` object for each call. Using this method is likely to have comparable cost to using the like-named field. (Unlike this method, the field does not provide type safety.)

Since:

1.5

See Also:

EMPTY_LIST

emptyMap

```
public static final <K,V> Map<K,V> emptyMap()
```

Returns the empty map (immutable). This map is serializable.

This example illustrates the type-safe way to obtain an empty set:

```
Map<String, Date> s = Collections.emptyMap();
```

Implementation note: Implementations of this method need not create a separate Map object for each call. Using this method is likely to have comparable cost to using the like-named field. (Unlike this method, the field does not provide type safety.)

Since:

1.5

See Also:

EMPTY_MAP

singleton

```
public static <T> Set<T> singleton(T o)
```

Returns an immutable set containing only the specified object. The returned set is serializable.

Parameters:

o - the sole object to be stored in the returned set.

Returns:

an immutable set containing only the specified object.

singletonList

```
public static <T> List<T> singletonList(T o)
```

Returns an immutable list containing only the specified object. The returned list is serializable.

Parameters:

o - the sole object to be stored in the returned list.

Returns:

an immutable list containing only the specified object.

Since:

1.3

singletonMap

```
public static <K,V> Map<K,V> singletonMap(K key,
                          V value)
```

Returns an immutable map, mapping only the specified key to the specified value. The returned map is serializable.

Parameters:

`key` - the sole key to be stored in the returned map.

`value` - the value to which the returned map maps `key`.

Returns:

an immutable map containing only the specified key-value mapping.

Since:

1.3

nCopies

```
public static <T> List<T> nCopies(int n,
                  T o)
```

Returns an immutable list consisting of n copies of the specified object. The newly allocated data object is tiny (it contains a single reference to the data object). This method is useful in combination with the `List.addAll` method to grow lists. The returned list is serializable.

Parameters:

n - the number of elements in the returned list.

o - the element to appear repeatedly in the returned list.

Returns:

an immutable list consisting of n copies of the specified object.

Throws:

`IllegalArgumentException` - if `n < 0`

See Also:

`List.addAll(Collection)`, `List.addAll(int, Collection)`

reverseOrder

```
public static <T> Comparator<T> reverseOrder()
```

Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the `Comparable` interface. (The natural ordering is the ordering imposed by the objects' own `compareTo` method.) This enables a simple idiom for sorting (or maintaining) collections (or arrays) of objects that implement the `Comparable` interface in reverse-natural-order. For example, suppose a is an array of strings. Then:

```
          Arrays.sort(a, Collections.reverseOrder());
```

sorts the array in reverse-lexicographic (alphabetical) order.

The returned comparator is serializable.

Returns:

A comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the `Comparable` interface.

See Also:

Comparable

reverseOrder

```
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

Returns a comparator that imposes the reverse ordering of the specified comparator. If the specified comparator is `null`, this method is equivalent to `reverseOrder()` (in other words, it returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface).

The returned comparator is serializable (assuming the specified comparator is also serializable or `null`).

Parameters:

`cmp` - a comparator who's ordering is to be reversed by the returned comparator or `null`

Returns:

A comparator that imposes the reverse ordering of the specified comparator.

Since:

1.5

enumeration

```
public static <T> Enumeration<T> enumeration(Collection<T> c)
```

Returns an enumeration over the specified collection. This provides interoperability with legacy APIs that require an enumeration as input.

Parameters:

`c` - the collection for which an enumeration is to be returned.

Returns:

an enumeration over the specified collection.

See Also:

`Enumeration`

list

```
public static <T> ArrayList<T> list(Enumeration<T> e)
```

Returns an array list containing the elements returned by the specified enumeration in the order they are returned by the enumeration. This method provides interoperability between legacy APIs that return enumerations and new APIs that require collections.

Parameters:

`e` - enumeration providing elements for the returned array list

Returns:

an array list containing the elements returned by the specified enumeration.

Since:

1.4

See Also:

`Enumeration`, `ArrayList`

frequency

```
public static int frequency(Collection<?> c,
            Object o)
```

Returns the number of elements in the specified collection equal to the specified object. More formally, returns the number of elements e in the collection such that `(o == null ? e == null : o.equals(e))`.

Parameters:

c - the collection in which to determine the frequency of o

o - the object whose frequency is to be determined

Throws:

`NullPointerException` - if c is null

Since:

1.5

disjoint

```
public static boolean disjoint(Collection<?> c1,
               Collection<?> c2)
```

Returns `true` if the two specified collections have no elements in common.

Care must be exercised if this method is used on collections that do not comply with the general contract for `Collection`. Implementations may elect to iterate over either collection and test for containment in the other collection (or to perform any equivalent computation). If either collection uses a nonstandard equality test (as does a `SortedSet` whose ordering is not *compatible with equals*, or the key set of an `IdentityHashMap`), both collections must use the same nonstandard equality test, or the result of this method is undefined.

Care must also be exercised when using collections that have restrictions on the elements that they may contain. Collection implementations are allowed to throw exceptions for any operation involving elements they deem ineligible. For absolute safety the specified collections should contain only elements which are eligible elements for both collections.

Note that it is permissible to pass the same collection in both parameters, in which case the method will return `true` if and only if the collection is empty.

Parameters:

`c1` - a collection

`c2` - a collection

Returns:

`true` if the two specified collections have no elements in common.

Throws:

`NullPointerException` - if either collection is `null`.

`NullPointerException` - if one collection contains a `null` element and `null` is not an eligible element for the other collection. (optional)

`ClassCastException` - if one collection contains an element that is of a type which is ineligible for the other collection. (optional)

Since:

1.5

addAll

```
@SafeVarargs
public static <T> boolean addAll(Collection<? super T> c,
                                 T... elements)
```

Adds all of the specified elements to the specified collection. Elements to be added may be specified individually or as an array. The behavior of this convenience method is identical to that of `c.addAll(Arrays.asList(elements))`, but this method is likely to run significantly faster under most implementations.

When elements are specified individually, this method provides a convenient way to add a few elements to an existing collection:

```
Collections.addAll(flavors, "Peaches 'n Plutonium", "Rocky Racoon");
```

Parameters:

`c` - the collection into which `elements` are to be inserted

`elements` - the elements to insert into `c`

Returns:

`true` if the collection changed as a result of the call

Throws:

`UnsupportedOperationException` - if `c` does not support the `add` operation

`NullPointerException` - if `elements` contains one or more null values and `c` does not permit null elements, or if `c` or `elements` are `null`

`IllegalArgumentException` - if some property of a value in `elements` prevents it from being added to `c`

Since:

1.5

See Also:

`Collection.addAll(Collection)`

newSetFromMap

```
public static <E> Set<E> newSetFromMap(Map<E,Boolean> map)
```

Returns a set backed by the specified map. The resulting set displays the same ordering, concurrency, and performance characteristics as the backing map. In essence, this factory method provides a `Set` implementation corresponding to any `Map` implementation. There is no need to use this method on a `Map` implementation that already has a corresponding `Set` implementation (such as `HashMap` or `TreeMap`).

Each method invocation on the set returned by this method results in exactly one method invocation on the backing map or its `keySet` view, with one exception. The `addAll` method is implemented as a sequence of `put` invocations on the backing map.

The specified map must be empty at the time this method is invoked, and should not be accessed directly after this method returns. These conditions are ensured if the map is created empty, passed directly to this method, and no reference to the map is retained, as illustrated in the following code fragment:

```
Set<Object> weakHashSet = Collections.newSetFromMap(
    new WeakHashMap<Object, Boolean>());
```

Parameters:

`map` - the backing map

Returns:

the set backed by the map

Throws:

IllegalArgumentException - if map is not empty

Since:

1.6

asLifoQueue

```
public static <T> Queue<T> asLifoQueue(Deque<T> deque)
```

Returns a view of a Deque as a Last-in-first-out (Lifo) Queue. Method add is mapped to push, remove is mapped to pop and so on. This view can be useful when you would like to use a method requiring a Queue but you need Lifo ordering.

Each method invocation on the queue returned by this method results in exactly one method invocation on the backing deque, with one exception. The addAll method is implemented as a sequence of addFirst invocations on the backing deque.

Parameters:

deque - the deque

Returns:

the queue

Since:

1.6